Self-Directed Lab Exercises

Writing WDF Drivers I:
Core Concepts

Designed and presented by:

OSR
open
systems
resources
inc.

Printed in the United States of America

Version: UL0523

LIMITED WARRANTY
OSR Open Systems Resources, Inc. (OSR) expressly disclaims any warranty for the material presented herein. This material is presented "as is" without warranty of any kind, either express or implied, including, without limitation, the implied warranties of merchantability or fitness for a particular purpose. The entire risk arising from the use of this material remains with you. OSR's entire liability and your exclusive remedy shall not exceed the price paid for this material. In no event shall OSR or its supplies be liable for any damages whatsoever (including, without limitation, damages for loss of business profit, business interruption, loss of business information, or any other pecuniary loss) arising out of the use or inability to use this information, even if OSR has been advised of the possibility of such damages. Because some states/jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

U.S. GOVERNMENT RESTRICTED RIGHTS
This material is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Right in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software--Restricted Rights 48 CFR 52.227-19, as applicable. Manufacturer is OSR Open Systems Resources, Inc. Manchester, New Hampshire 03031.

Contents

OSR’s GitHub Repos 5

Contents of the WDF-I Repo 5

About the Lab Exercises – PLEASE READ 6

How the Labs Are Organized 6

Seminar Source Code Samples and Test Programs 7

UMDF 2, Anyone? 7

The VMware Virtual Machine Image 8

A Quick Note on Using WinDbg 9

Working with Visual Studio and the Driver Samples 9

Start Here! Do This First to Avoid Potential Annoyance! 13

Preparations for Beginning the Lab Assignments 13

Lab Exercise 1 – Building, Debugging, and Installing 14

Assignment 1A 15

Assignment 1B 15

Assignment 1C 16

Assignment 1D 18

Assignment 1E 19

Lab Exercise 2 – Driver Initialization 21

Assignment 2A 21

Assignment 2B 21

Lab Exercise 3 – Requests, Buffering, and Queuing 23

Assignment 3A 23

Assignment 3B 23

Lab Exercise 4 – Filter Drivers 25

Assignment 4A 25

Assignment 4B 25

Assignment 4C 25

Lab Exercise 5 – USB 27

Assignment 5A 28

Assignment 5B 28

Assignment 5C 29

Lab Exercise 6 – File Objects, Open and Close 31

Assignment 6A 31

Setting Up a VMWare Virtual Machine for Use with WinDbg 33

Introduction 33

Steps 33

Using the OSR USB FX-2 Learning Kit V2.0 45

Configurations 45

Interfaces 45

Endpoints 45

Endpoint 1 -- Interrupt Endpoint Packet Format 46

Endpoint 6 and 8 Bulk Packet Format 46

Board LED Displays 47

7-Segment LED display 47

LED Bar Graph Display 47

Vendor Commands 47

Advanced Usage Information 50

Device Problems or Questions 50

Frequently Asked Questions 51

Differences Between OSR USBFX-2 V1.0 and V2.0 51

OSR Thanks 52

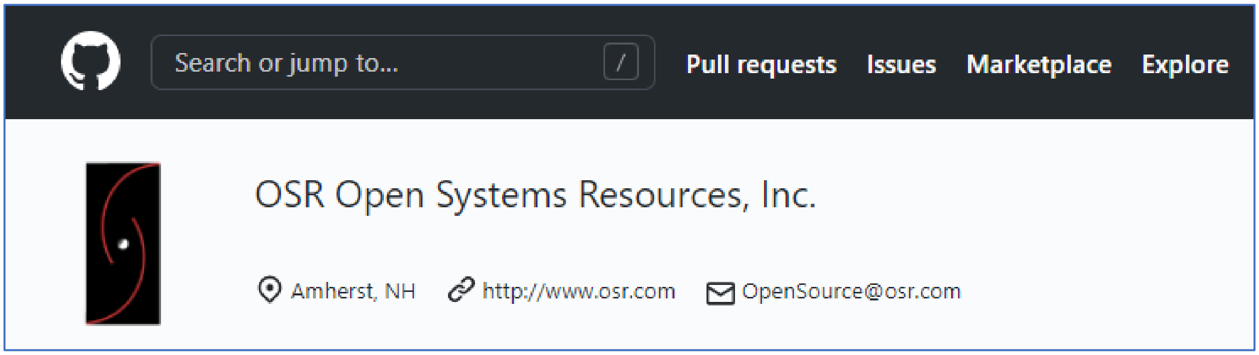
OSR's GitHub Repos

OSR provides a set of repositories on GitHub that contain sample code that we think might be interesting or useful to members of the community. You will find our repos at:

https://github.com/OSRDrivers

The samples that are specific to this seminar are in the **WDF-I** Repository. We describe the contents of this repo in more detail below and in the following pages.

We recommend you check our GitHub page regularly, as we regularly updated and add to the projects we share with the community.



Contents of the WDF-I Repo

Please download the entire contents of the WDF-I repo to your development system (the system on which you'll be running Visual Studio and VMware).

At the time this handout was written, it contained the following folders:

Directory	Contents
\BasicUSB \CDFilter \Nothing_KMDF	Code samples for lab exercises (described in more detail later)
\Docs_and_Misc	This (Lab Exercises) document; Other items including the Empty ISO image that can be used as part of the CD Filtering lab assignment.
\Solutions	Lab exercise solutions (source code); Organized by exercise number

About the Lab Exercises – PLEASE READ

These lab exercises are designed for use by students who take OSR’s Windows Writing WDF Drivers seminar. We don’t expect them to make sense outside of that context. Despite that, we have put the exercises on GitHub – along with the sample drivers to which these exercises refer – and if people who haven’t yet taken our WDF seminar find them helpful, that’s great.

Because the learning styles and goals of each seminar student is different, these lab exercises are designed as an optional part of the seminar. They are also intended to be “self-directed.” That is, we expect the lab exercises will be done by a student on their own, outside of class. During your seminar, the instructional team will be happy to answer any questions you might have about any of the lab exercises. After the conclusion of your seminar, you may still feel free to email your instructor at any time with any questions you might have.

How the Labs Are Organized

Lab work has been divided into individual numbered exercises, each of which comprises several individual assignments. It’s generally a good idea to **read through all the assignments** within a particular exercise **before** starting to work on one of the assignments.

The assignments are generally meant to be done in order, and progress in terms of difficulty. Later assignments (and exercises) often build on the results of previous assignments. Choose an assignment that sounds interesting to you and work on that. Start simple. Anything that requires you to actually get code to work will almost certainly take somewhat longer than you anticipate. Sometimes it will take **much** longer. For example, it took one of the OSR instructors over five hours to solve one of the simpler assignments (he says it was a “mental block” … the rest of us aren’t so sure)!

For each assignment, we list the sample driver that we recommend you start from, as well as a series of hints for completing the assignment. In many cases, the hints really tend to give the answer away. Therefore, **first try to do the assignment <u>without reading any of the hints</u>!** If you do need a hint, try reading the hints one at a time.

Seminar Source Code Samples and Test Programs

OSR provides a set of driver source code to help you get started with the lab exercises. We recommend that you start from these drivers when doing the exercises. The sample drivers for this seminar are stored in the **WDF-I** repo on the **OSRDrivers** page on GitHub. You can either clone the WDF-1 repo using Git or download the contents of the repo as a ZIP archive.

We provide three, specific, sample KMDF drivers with this Seminar. Each driver is provided with a working INF file that installs the sample. The samples are:

- **Nothing Driver** – Located in the **Nothing_KMDF** directory. This is a solution that contains a project with that builds a KMDF software driver that does nothing much. The driver implements Event Processing Callbacks for read (completes all requests received with success with zero bytes of data returned), write (completes all requests received with success indicating all requested bytes written), and device control (completes all requests received with success and zero bytes of data returned). This driver is a great starting point for most of the labs (AND a lot of drivers in "real life" after class).

  Also included in the solution is a utility named **nothingtest** that can be used to open and send reads, writes, and device controls to the nothing driver.

- **Filter Driver** -- Located in the **CDFilter** directory. This a solution containing a driver project that implements the bare skeleton of a generic KMDF filter driver. This is the sample driver you'll use if you choose to do a lab about filtering.

- **Basic USB Sample Driver** – Located in the **BasicUsb** directory. This is a simple KMDF driver that provides minimal support for the OSR USB FX2 device. It currently only contains support for bulk write, a continuous reader that prints the result of a switch pack change, and an IOCTL accessible vendor command that lights the bar graph.

  Also included in the solution is a utility named **basicusbtest** that can be used to open and send requests to the **basicusb** driver.

Solutions to each of the lab exercises, organized by lab exercise number, are in the **Solutions** directory of the repo.

UMDF 2, Anyone?

We have designed the Nothing Driver and the Basic USB Sample Driver to work as either KMDF or UMDF 2 drivers. While we provide the Visual Studio and WDK Solutions for KMDF, you can make them work in UMDF 2 by:

1. Creating a new directory for the UMDF driver to use.
2. Copying all the .CPP, .H and .INF files to that new directory.
3. Creating an empty UMDF Project (and Solution) that references that new directory.
4. Adding the sources and headers that you copied (in step 2) to the UMDF Project.
5. Bingo! You should have a working UMDF version of the sample driver.

The VMware Virtual Machine Image

We provide a VMware Windows image that you can use to do the lab exercises. This image is typically supplied by the OSR Seminar Team on a USB flash drive, but for some seminars a link to download this image may be provided. This image is just a VMware virtual machine with an empty Windows system installed. And it’s usually not a very recent version of Windows, either. The purpose of this VM image is to give you a quick way to create your Target system, which is the system on which you will test your drivers you write during these lab exercises.

Please be aware that there’s nothing “special” about this image: It’s just an empty Windows VM, but using it will allow you to follow the step-by-step setup instructions we provide in this handout. If you’re already familiar with the basics of setting up a Target system for debugging with WinDbg, feel free to use a physical machine or a VM of your own creation.

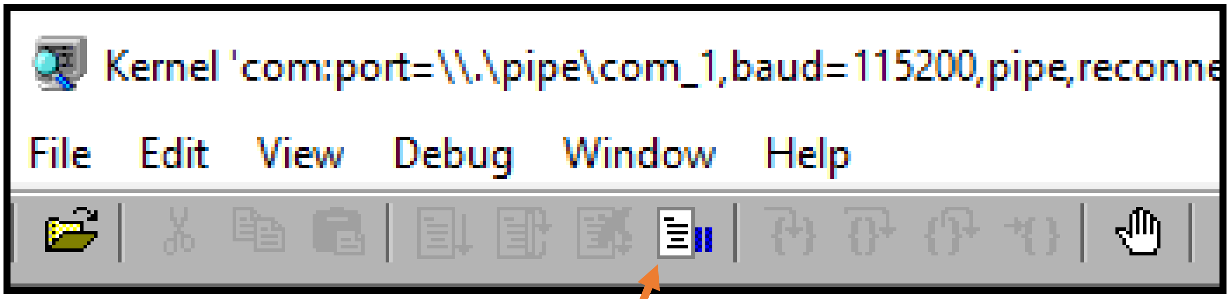
Also, please note that the VM has “host only” networking enabled. That allows the VM to communicate with your host machine, but NOT to access the Internet. Do not enable Internet access from this VM. You absolutely do not want Windows Update to start running on your VM while you’re trying to test and debug your drivers.

The username for the Windows system on the VM is Student, and the password is STUDENT (all in capitals)

A Quick Note on Using WinDbg

When you're using WinDbg (or any kernel-mode debugger), you must STOP the Target system before you can enter any commands to examine the target system's memory, set a dynamic breakpoint, or do any other control operation on the Target. Stopping the Target allows WinDbg to take control of the Target machine.

To STOP the target system, use the STOP button on WinDbg's menu bar:


Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe,reconne

This is the "STOP" button. It causes the Target machine to pause, and allows you to enter WinDbg command-line commands to examine memory in or

Working with Visual Studio and the Driver Samples

When working with Visual Studio (VS), there are a few things that you need to keep in mind. First, you need to keep in mind that VS groups program code into “Projects” and “Solutions.” A **Project** (file type .vcxproj) is a collection of source code files, private headers, resources, references to other items (public headers, libraries, and/or other resources), and the instructions necessary to build these things together into a single component. For example, a Project might be all the files and build instructions necessary to create an application program or a driver or a dynamic link library. Interestingly enough, Projects can also be created that do not contain any program code at all – You can create a Project that includes only steps to copy, rename, or process certain files, for example.

A VS **Solution** is a container that groups together one or more projects. VS Solution files have the file type .sln. A Solution allows you to easily group multiple Projects that you might want to build at the same time. Optionally, Projects within a Solution can have specific dependencies on other Projects within the same solution. This is the case, for example, when you need to build an application program that uses features provided by a DLL that’s part of the same Solution. When a Solution groups multiple Projects together, the Solution is placed in a higher-level directory and each Project that is part of the Solution is usually stored in a sub-directory of the Solution.

Each of the sample drivers we provide is organized as a Solution with multiple Projects. When working with the driver samples and Visual Studio, it is often easiest to use the file menu item in VS, select the Open… item, and select Project/Solution. This dialog will allow you to navigate to the sample Solution that you want to use.

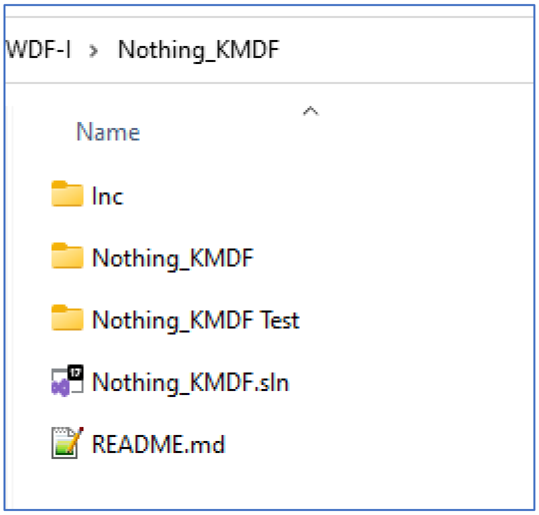
Note that both Solutions and Projects have “Properties.” You can review and set these Properties by right clicking the Solution or Project (in the VS Solution Explorer window) and selecting Properties. Note that unless you specifically select “All Configurations” and “All Platforms”, the **Properties you set on a Project are specific to the Configuration (Debug/Release) and Platform (x64 or ARM)** that is selected. You probably won’t notice this during class because you’ll always be building for just one Configuration and Platform in lab. However, keep in mind that if you set a Property (such as the compiler warning level, the level of Code Analysis you want to use, or adding a specific LIB to link against) for one Configuration and Platform (let’s say for Debug and x64), you’ll need to set that Property in *every* Configuration where you want that setting to apply.

When building within VS, you can choose to build either an entire Solution or an individual Project. Building a Solution builds all the Projects within that solution, and in a specific order. Building a Project only builds the selected Project.

While you are actively developing your driver in class you’ll want to build (or re-build) the Project that contains your driver. However, once you are ready to begin testing your driver, we strongly recommend that you **rebuild the entire Solution** (using the “Rebuild Solution” menu option from the “Build” menu item in VS). Rebuilding the Solution causes all elements of all projects to be built from scratch. Note that driver projects automatically create a neat and tidy “package” directory (that the same name as the project) with all the driver components in it. You can copy the contents of this directory to your test system for testing.

Above all, keep in mind that the VS Project files contain all the settings for the component to be built. The Project includes information about source files locations, the Project include paths, and where the resulting images should be placed.

When you build a driver project, the build outputs are placed in a directory at the top-level of your solution, based on target architecture. Under THAT directory, you’ll find Debug and Release directories with all the build products. And inside THAT directory you’ll the driver “package” directory described above. So, for example, let’s look at the **Nothing_KMDF** driver sample driver. When you clone the repo from GitHub, the top-level directory will look something like this:

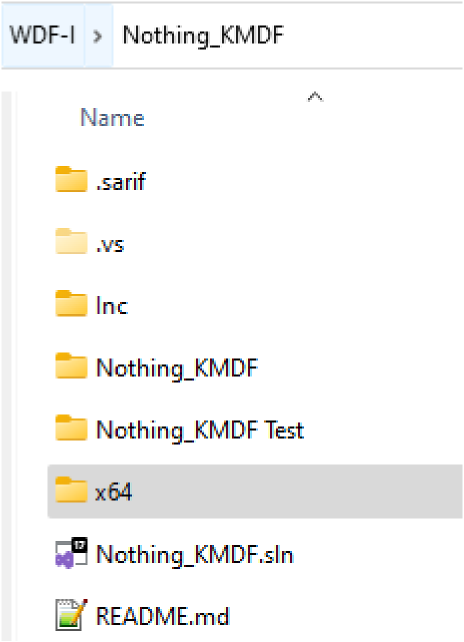


WDF-I › Nothing_KMDF

Name

Inc
Nothing_KMDF
Nothing_KMDF Test
Nothing_KMDF.sln
README.md

When you build the Solution, a set of directories will be created with “stuff” from Visual Studio (these directories start with a dot), as well as the build products:

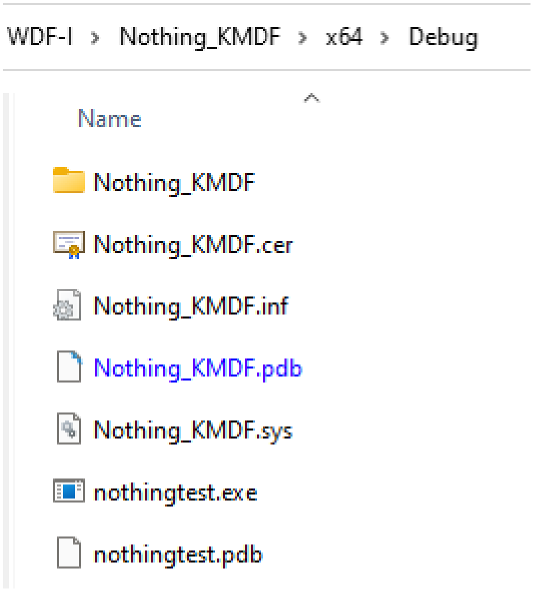


WDF-I › Nothing_KMDF

Name

.sarif
.vs
Inc
Nothing_KMDF
Nothing_KMDF Test
x64
Nothing_KMDF.sln
README.md

Notice the x64 directory? Under that, you'll find Debug and Release directories (when you build Debug and Release builds of your code). And in the Debug directory (for example), you'll find the actual build products:

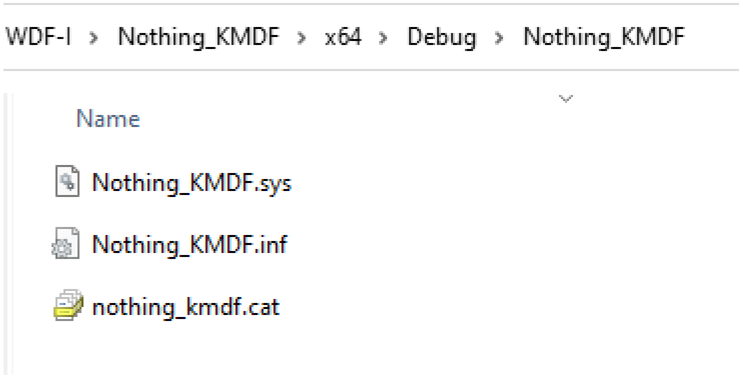


WDF-I › Nothing_KMDF › x64 › Debug

Name

Nothing_KMDF
Nothing_KMDF.cer
Nothing_KMDF.inf
Nothing_KMDF.pdb
Nothing_KMDF.sys
nothingtest.exe
nothingtest.pdb

This includes the driver executable (Nothing_KMDF.SYS), the installation control file (Nothing_KMDF.inf) and a copy of the certificate used to sign the driver executable and package (Nothing_KMDF.cer). This directory also contains a directory with the project name (Nothing_KMDF in the example):



WDF-I › Nothing_KMDF › x64 › Debug › Nothing_KMDF

Name

Nothing_KMDF.sys
Nothing_KMDF.inf
nothing_kmdf.cat

This is the driver "package" directory mentioned previously. You see it has in it a copy of the driver's SYS file, the INF file, and the driver's security catalog file (nothing_kmdf.cat) that was built (and signed) when you built your driver. The INF file refers to the CAT file, so you'll need to copy it to your test system when you want to install your driver.

Start Here! Do This First to Avoid Potential Annoyance!

Preparations for Beginning the Lab Assignments

Before beginning the labs there are some things you need to do. These steps are necessary to ensure that the installation of Visual Studio (VS) and the WDK are correct and that everything works as expected. **Please do not start work on any of the lab assignments before you’ve completed these steps**.

1. Start Visual Studio (VS). In the “Get Started” dialog select “Create a new Project”. A “Create a new project” dialog should appear where there should be entries for different driver types (“Filter Driver”, “Kernel Mode Driver”, etc). If this is what you see, “Cancel” out of the dialog and exit VS (click the X in the upper right corner) and proceed to Step 2. Otherwise ask your instructor to look at your system installation before you do anything else.

2. Ensure that the Windows Debugger (WinDbg) is installed on your development system and that you can start it. You should find WinDbg in the directory %ProgramFiles%\Windows Kits\10\Debuggers\x86\ -- You might want to put a short-cut in your tool tray to the file in this directory. Note that depending on how you installed the SDK, you may also have an entry for WinDbg in your Start screen. This will point to the version installed by the SDK.

Once you have done these steps proceed to lab 1.

(This page intentionally left blank)

Lab Exercise 1 – Building, Debugging, and Installing

These exercises all have to do with installing and using the available tools to compile, link, install, and debug drivers.

Assignment 1A

If you have not already done so, copy the contents of the **WDF-I** repo (described previously… see the sections entitled *OSR’s GitHub Repos* and *Seminar Source Code Samples and Test Programs* to a dedicated directory (e.g., C:\WDFDRIVERLAB) on your Development system. **Change the file attributes of all the files and directories that you copy to allow Read/Write access.**

Also, copy the VMware VM image to your development system. This image is typically supplied by the OSR Seminar Team on a USB flash drive, but for some seminars a link to download this image may be provided. This image is just a VMware virtual machine with an empty Windows system installed. It will be used as your Target system, on which you will test your drivers you write during these lab exercises. **The username for the Windows system on the VM is Student (case doesn’t matter in Windows usernames), and the password is STUDENT (all in capitals, Windows passwords ARE case sensitive)**

Assignment 1B

Set up for WinDbg driver debugging on your systems. Your laptop, containing VS and the WDK, will be your Host/Development system and the virtual machine will be your Target system. The roles of these two systems won't change during the seminar.

Proceed as follows:

1. For your maximum convenience, we recommend that you pin the Command Prompt Window icon to your system’s task bar on your Target (test) system and that you pin both VS and WinDbg icons to the task bar on your Development (host) System. It’s up to you, but this WILL make your life easier during lab this week.

2. If your Target is a VM you are going to need a Serial Port that can be used for the Development System and the VM to communicate with. If the VM doesn't already have a serial port available for use, you'll need to modify the settings of your Virtual Machine to add the serial port. Be sure that when you create it, you tell your VM that it should create a “named pipe” and keep the defaults of “This end is the server.” and “The other end is an application.” The name that you give the named pipe will be needed by WinDbg to connect to your VM, for example: **\\.\pipe\com_1**. In some version of VMware, to be able to change the serial port characteristics you need to enable “virtual printer” support. To do this, in VMware select “Edit”, then “Preferences…”. Next, in the Preferences dialog box, select “Devices” and check the “Enable virtual printers” check box. It’s harder to describe than it is to do.

   If you’ve never set up a VMWare Virtual Machine for debugging before, you will probably want to refer to the class handouts or the section “Setting Up a VMWare Virtual Machine for Use with WinDbg” later in this document. This section will walk you through the process of setting up the VM and also setting up WinDbg to connect to the VM (which is also discussed in steps 3 and 4 below).

3. If you haven't already done so (by following the steps in the "Setting Up a Virtual Machine" walk-through, for example) enable kernel debugging on your Target system:

    - Use the BCDEDIT utility (remember to "run as administrator") to enable debugging:

        ```
        bcdedit /debug on
        ```

    - Change the default debugger connection settings if necessary. The defaults settings are usually acceptable, but they are:

        ```
        bcdedit /dbgsettings serial debugport:1 baudrate:115200
        ```

4. Again, if you have not already done so, setup WinDbg on the Development (host) system. Start WinDbg and select "Kernel Debug…" from the "File" menu. If your target is a VM then WinDbg will need to connect to your VM via a pipe, specifying the name of the named pipe in the "Port:" field of the COM tab of the Kernel Debugging dialog box. If your target is not a VM, then specify the port number (such as "com1") of the serial port on your Development system in the "Port:" field of the COM tab of WinDbg's Kernel Debugging dialog box.

    When settings in the Kernel Debugging dialog are complete, click OK. Ensure WinDbg is “waiting for connection” in the command Windows.

    Next, reboot the Target system and ensure you can make a connection between your host (dev) system and your target system (WinDbg will display some information about the O/S on the Target system as soon as the connection is established).

5. Be sure to set up the symbol search path correctly in WinDbg. The public Microsoft symbol server is located at:

    ```
    srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
    ```

    Optionally you can avoid typing the above long path specification and set the default symbol path using the WinDbg command .SYMFIX.

    After you’ve setup the symbol search path, tell WinDbg to reload its symbols using the WinDbg command:

    ```
    .RELOAD
    ```

Assignment 1C

Build (on your Development system) and install (on your Target system) a “Debug” and “x64” version of the Nothing_KMDF sample driver as follows:

- Open the Nothing_KMDF.sln file using VS (you can just double-click the .SLN file) and build the solution. Be sure to build the whole solution, not just the driver project!

- The architecture built (x64 or ARM64) **must** match the architecture of the target system. Use WinDbg to determine the target architecture if you’re not sure, it is displayed as part of the initial connection information.

- Create a dedicated installation directory on the Target. For example, for the nothing driver you might create the directory c:\nothing

- Copy the WDK’s DevCon utility (from the WDK's %ProgramFiles%\Windows Kits\10\tools\x64 directory) to your dedicated installation directory on the target. The architecture of DevCon (x86 or x64) **must** match the architecture of the target system.

- Copy the contents of the created driver package directory which includes the driver-related files you’ll need to copy from your Development system to a dedicated installation directory on the target. These will be in the “x64\ Debug\Nothing_KMDF” subdirectory under your Nothing_KMDF Solution directory, depending on the target architecture you chose (x64 in this example). The following three files will be in the Package sub-directory:

  - Nothing_kmdf.cat
  - Nothing_KMDF.inf
  - Nothing_KMDF.sys

- Start an administrator command prompt (right click on the command prompt icon and select “Run as Administrator”) Window on the target system

- Be sure to enable output from DbgPrint/DbgPrintEx on your Target system! You can do this with the WinDbg command:

  ```
  ed Kd_DEFAULT_MASK 0xF
  ```

  If your symbol settings in WinDbg are not correct, this command will not work. Consider this a good opportunity to get your symbols set up correctly. Note you’ll have to issue this command EVERY TIME your reboot your Target system to enable output from DbgPrint.

  OR, even better, on your Target system set the following value in the Registry:

Path:	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter
Name:	DEFAULT
Type:	REG_DWORD
Value:	0xF

  You will almost certainly need to create the “Debug Print Filter” key. Note that the value name must be “DEFAULT” (without the quotation marks) – This isn’t the same as the “(default)” value for the key that’s created automatically in the Registry.

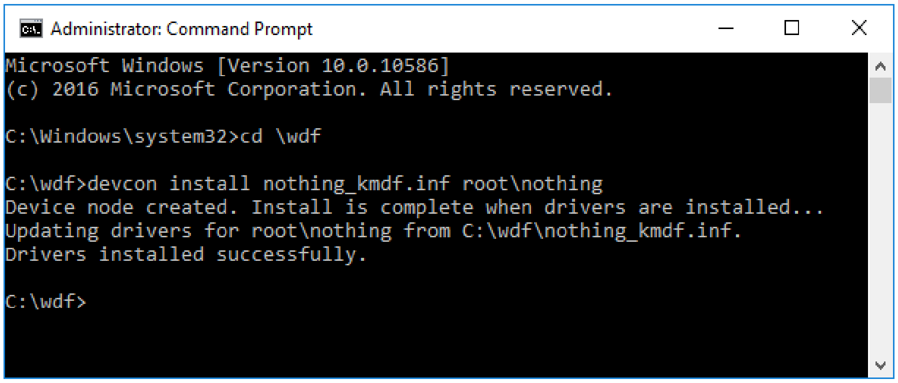
  If you choose to set the Registry value, you will have to reboot the Target for the change to take effect, but this setting will remain in effect for each reboot. In other words, if you make the Registry change, you will NOT have to manually enable DbgPrint output each time you reboot your system.

- From the (elevated) command-prompt window on the Target, use the DevCon utility from the WDK to install your driver. For software-only drivers (such as nothing) use the “DevCon install” command. Note the parameters to DevCon. It requires the hardware id of the device, which can be determined from the .inf file. So, the command to install the nothing driver is:

"Devcon Install Nothing_KMDF.inf ROOT\Nothing"

(Case is not important in the above command line)

Read the messages displayed during installation carefully. If there were no errors, you should see debug output in the WinDbg command window when the "Nothing_KMDF Driver" is loaded.



```
Microsoft Windows [Version 10.0.10586]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Windows\system32>cd \wdf

C:\wdf>devcon install nothing_kmdf.inf root\nothing
Device node created. Install is complete when drivers are installed...
Updating drivers for root\nothing from C:\wdf\nothing_kmdf.inf.
Drivers installed successfully.

C:\wdf>
```

Check with Device Manager to ensure your driver is loaded: You should see a Nothing class of devices with the Nothing device loaded and working.

If the installation failed for some reason, check your commands carefully. If you're stumped as to the cause, you can check the Windows installation log to see exactly what happened. That file is:

C:\Windows\INF\setupapi.dev.log

Yes… that's ".dev.log" at the end. Start at the bottom of this file and work backwards. The number of exclamation point starting on the left margin indicates the severity of any issues that are encountered. In other words, a message starting with "!" is less important or severe than one that starts with "!!!" – Pretty clever, huh?

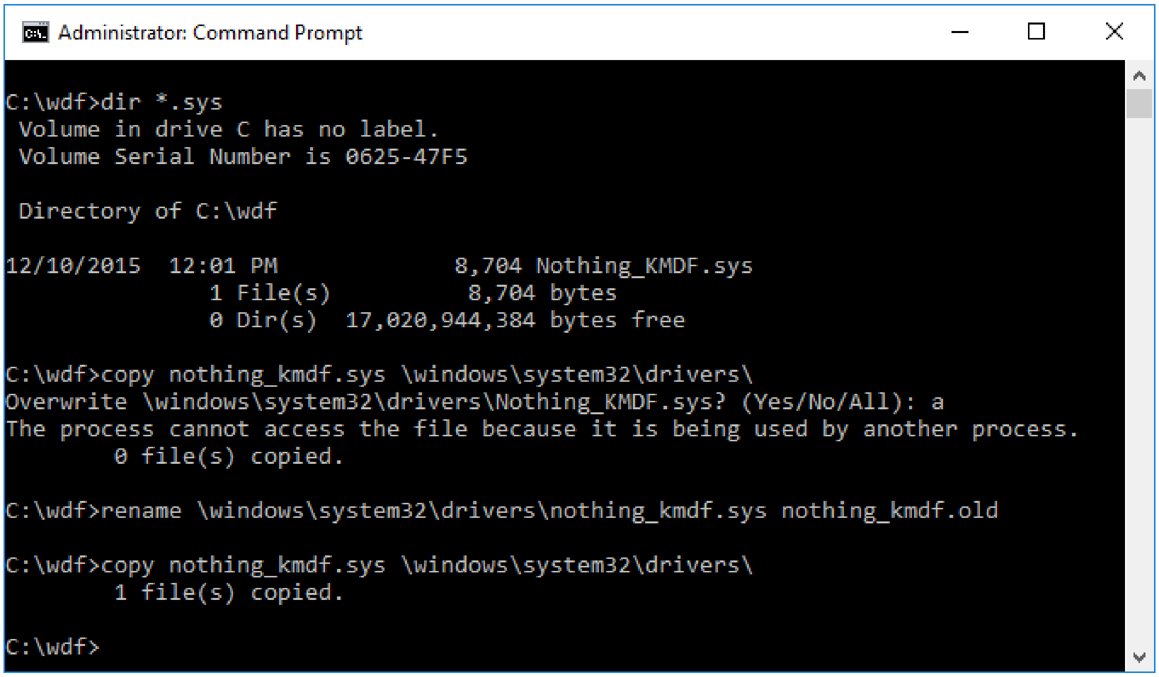
Assignment 1D

Before you do this assignment make sure that you can execute the "!wdfkd.help" command in WinDbg. If not, configure WinDbg to use the WDF Kernel Debugger Extensions by loading the extension wdfkd.dll (!load wdfkd.dll). "!wdfkd.help" needs to work before you proceed.

Dump the WDF Log (otherwise known as the In Flight Recorder or IFR) using the command "!wdflogdump nothing_kmdf". The output should contain some semi-useful information about your driver.

Assignment 1E

Place a few DbgPrint statements in the DriverEntry and EvtDriverDeviceAdd functions of the Nothing_KMDF Driver. Then, rebuild the driver and **copy it to your Target System's dedicated directory.** <u>Do not re-install the updated version of the driver using DevCon</u>. Instead, from an elevated command prompt copy the updated version of your driver to the \Windows\System32\Drivers directory of your Target System. **If your Target System is running Windows 10 or later**, you will first rename the existing driver (so "rename \Windows\System32\Drivers\Nothing_KMDF.sys Nothing_KMDF.old" or something similar) and then, after the previous version has been renamed, you can copy the updated version of your driver to the \Windows\System32\Drivers directory.



```
C:\wdf>dir *.sys
 Volume in drive C has no label.
 Volume Serial Number is 0625-47F5

 Directory of C:\wdf

12/10/2015  12:01 PM             8,704 Nothing_KMDF.sys
               1 File(s)          8,704 bytes
               0 Dir(s)  17,020,944,384 bytes free

C:\wdf>copy nothing_kmdf.sys \windows\system32\drivers\
Overwrite \windows\system32\drivers\Nothing_KMDF.sys? (Yes/No/All): a
The process cannot access the file because it is being used by another process.
        0 file(s) copied.

C:\wdf>rename \windows\system32\drivers\nothing_kmdf.sys nothing_kmdf.old

C:\wdf>copy nothing_kmdf.sys \windows\system32\drivers\
        1 file(s) copied.

C:\wdf>
```

After you've copied the updated version of the driver to the \Windows\System32\Drivers directory, load the new copy of the driver and check that your changes worked. Do this by disabling and then re-enabling the Nothing device using Device Manager, or "restarting" the driver using devcon.

Note that the process just described is the standard process for updating a driver once it's been installed (both here in class and back at your office when your debugging). To review, that process is:

1. Copy the new image to \Windows\System32\Drivers (renaming the previous version first if you're target is Windows 10 or later) and then

2. Disable and re-enable the driver using Device Manager or devcon.

There are other clever ways to do this as well. Google around (or ask your instructor to tell you) about using "kdfiles" – it's pretty nice, and if you learn to use ".kdfiles -m *<oldriver> <full-path-to-new-driver>*" you will avoid most of the annoyance that has historically turned folks away from using this command.

Next, add code to the KMDF_Nothing driver’s DriverEntry function to call the function DbgBreakPoint(). When this function call is executed, your driver will breakpoint in the debugger. Again, restart the driver (whichever way you prefer) to test your change.

Finally, add one or two dynamic breakpoints using WinDbg and the driver sources. To do this, you’ll need to stop the target machine (hit the pause button in WinDbg’s menu bar), select the line in WinDbg’s source code window where you want the break point to happen, and set the breakpoint by hitting the breakpoint button in WinDbg’s menu bar. When the break point is set, the selected line will change color. Practice using WinDbg to single step from your breakpoints; examine values for variables and structures, etc.

Lab Exercise 2 – Driver Initialization

These exercises are designed to help you become familiar with the various driver initialization functions, while giving you a bit more practice using the WDK and WinDbg.

For these assignments, use the version of the Nothing_KMDF driver that you modified in Lab 1.

Assignment 2A

Examine the Nothing driver's source code. Notice that it creates an unnamed Device Object and a Symbolic Link, but it does not create a device interface.

Modify the Nothing driver to create a device interface for its Device Object with the interface class GUID_DEVINTERFACE_NOTHING. Note that this GUID is already defined in the file nothing.h.

After you've made your changes, you can use the NothingTest utility to attempt to open the nothing device using the device interface (instead of by device name). To do this, run NothingTest with any parameter on the command line (e.g., " NothingTest –i") -- Supplying any parameter will cause NothingTest to attempt to open the nothing device by device interface instead of by name. Refer to the sources for NothingTest for more information.

Assignment 2B

Add EvtDeviceD0Entry and EvtDeviceD0Exit Event Processing Callbacks to the nothing driver. Be sure to put DbgPrint calls in these functions so that you know when they execute. Note when they're called. Is EvtDeviceD0Exit called when the nothing driver is disabled? How about during system shutdown?

(This page intentionally left blank)

Lab Exercise 3 – Requests, Buffering, and Queuing

These exercises provide practice manipulating data to satisfy read and write requests, as well as queuing Requests and dealing with Completion Routines.

Once again, the Nothing_KMDF driver is a good starting point for these exercises.

Assignment 3A

Create a driver that internally buffers the data from Write requests and uses that data to satisfy subsequent Read requests.

The driver should buffer the data from only one Write at a time (in other words, there’s no need to store the data from multiple write requests simultaneously). If the driver gets a read, but there’s no waiting Write data buffered by the driver, complete the Read with success returning zero data bytes. If the driver gets a Write, but there’s already data buffered that hasn’t yet been returned to a Read, complete the Write with an error.

Again, to test your driver, you can use the nothingtest utility, which is already setup to support both read and write operations.

Hints:

1. Complete both Reads and Writes synchronously. That is, complete each Read and Write you receive directly within its I/O Event Processing Callback. Don’t get fancy; you have lab assignment 3B for that!

2. You’ll need to impose a maximum size on the Write data the driver buffers – There are several reasonable choices for where to store the buffered Write data, depending on the maximum Write size you choose.

3. You can allocate space for buffering the data in your Device Context structure. Or, alternatively, use WdfMemoryCreate to allocate paged or non-paged scratch storage space from your driver.

Assignment 3B

Expanding on the concept of the driver described in Assignment 3A, create a driver that uses the data from Read requests it receives to satisfy the Write requests. For this assignment, queue both Read and Write Requests until they can be satisfied. Thus, for example, when the driver receives a Read request, it checks its queue of previously received Write Requests. If no Write Requests are queued, the driver queues the Read Request and leaves it pending. However, if a Write Request **is** present on the driver’s queue of previously received Write Requests, the driver uses the data from that queued Write to satisfy the Read (transferring data between the buffers) and completes both requests.

Similarly, when the driver receives a Write request, it checks to see if a Read has been queued. If it has, the driver uses the Read from its internal queue to satisfy the Write and completes both requests. If there is no Read queued when the Write is received, the driver queues the Write request and leaves it pending.

To test your work, you can use the nothingtest utility. But, you'll notice that we were lazy when we wrote nothingtest: It sends both reads and writes synchronously (that is, it waits for the call to ReadFile and WriteFile to complete before it will process another command). As a result, to test your driver for this lab exercise, you'll need to run two copies of nothingtest: One to send the read (which will wait until the read is complete), and another to send the write that'll be used to satisfy that read.

This is a relatively challenging assignment.

Hints

1. You'll need to maintain two queues with DispatchType set to WdfDispatchTypeManual in your driver: A queue for Read Requests and a queue for Write Requests.

2. Think about the DispatchType required for your device's default queue. For this exercise, your driver needs to be able to have multiple simultaneous requests outstanding from the default queue, right? Does that mean you need to change the dispatch type to something other than Sequential?

3. Because you're leaving requests pending in your driver, your driver will need to deal with the case where the user-mode requestor exits with requests in your queue. That means you're going to have to consider request cancellation. Are there any issues to worry about? Don't worry about solving this problem yet, just consider the problem. We'll talk about Request cancellation later in the class.

Note: After you've finished Assignment 3B, keep the sources for this driver around. You'll use it again as part of Assignment 6A... if you get there before the end of the week, that is!

Lab Exercise 4 – Filter Drivers

In these assignments, you'll write a filter driver for the CD-ROM device on your Target system. For this lab exercise, we recommend that you start with the CDFilter driver sample that we provide.

NOTE: You'll need to generate some CD-ROM activity to see requests in your CD-ROM filter. For that purpose, we have provided a (blank) ISO image that you can mount in your guest VM. You will find this empty ISO image in the \Docs_and_Misc directory of the WDF-I GitHub repo associated with this seminar. For more details on the GitHub Repo see the earlier section of this handout entitled OSR's GitHub Repos.

Assignment 4A

Build the provided CDFilter sample driver and install it on your Target system. To install the CDFilter, copy both the .SYS and .INF files created by the CDFilter solution to a dedicated directory on your Target machine. To install the filter, navigate to that directory in Explorer and right-click on the INF file and select "Install." You will NOT see the filter in Device Manager, but you should see your DbgPrint output in WinDbg (assuming DbgPrint output is enabled as was discussed in Lab Exercise 1).

To initially load or to update your filter, you will need to disable then enable the CD-ROM device using Device Manager (just right-click the CD-ROM in Device Manager and select "Disable", etc) or reboot the machine.

Note that the CDFilter sample driver doesn't do much in the form we provide it to you. It DOES however DbgPrint messages from its DriverEntry and EvtDriverDeviceAdd Event Processing Callback. Be sure you have DbgPrint output enabled on the Target machine and be sure you see the CDFilter load when a CDROM is mounted before going on to the next part of this lab assignment.

Assignment 4B

Modify the provided sample to display the number of bytes requested by each Read Request it receives. Don't get fancy (yet)... We suggest you use SEND_AND_FORGET to send the Read Request to your Local I/O Target when you're done with it.

Be sure your filter installs correctly, and still allows the CD-ROM to function correctly.

Hints:

1. Don't forget to tell the Framework that your driver is a filter driver.

2. You only need to create an Event Processing Callback for Requests you want to examine. The Framework will automagically forward all other Requests for you.

Assignment 4C

Modify the filter driver that you created in the previous assignment to display the completion status (including both status and additional information) of each Read request, after it has been processed by your driver's Local I/O Target.

Again, be sure that your driver doesn't "break" the CD-ROM stack on your target system.

Hints:

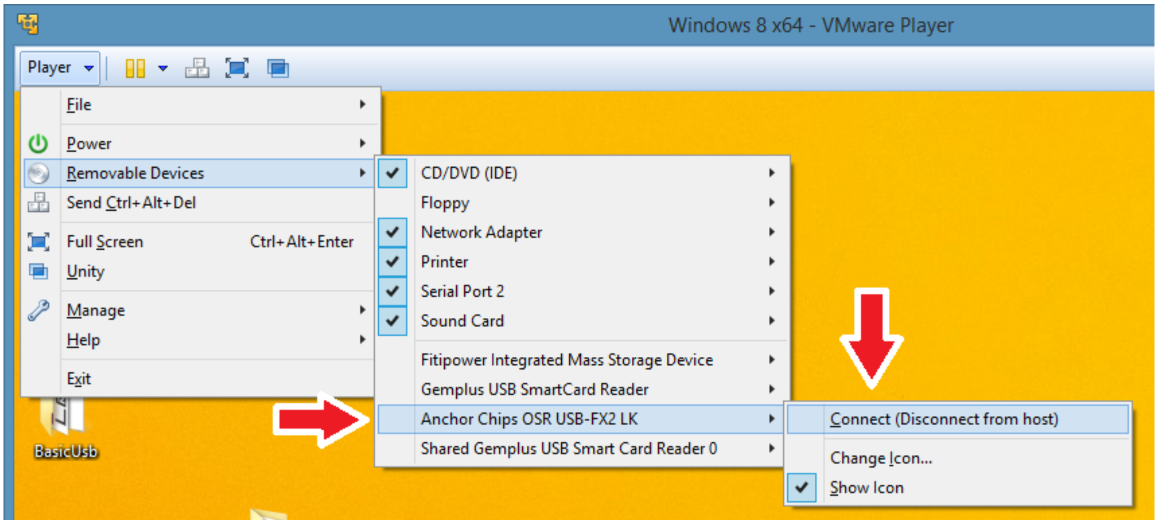
1. There are several ways that you could implement this assignment. We'd suggest the easiest and most useful is probably using a Completion Routine.

2. Remember that your driver must call WdfRequestComplete for every Request it receives, and that it doesn't send using SEND_AND_FORGET.

Lab Exercise 5 – USB

The following lab exercises use the Basic USB driver as a base of exploration into the fundamental operations of a KMDF USB driver. The Basic USB driver is a function driver for the OSR USB FX2 device, thus you must have an OSR USB FX2 device to perform these lab exercises.

In the VM environment, the OSR USB FX2 must first be given to the VM. To do this, plug the USB device into your system as you normally would and then instruct VMware to give the device to the guest:


Windows 8 x64 - VMware Player
Player
File
Power
Removable Devices
Send Ctrl+Alt+Del
Full Screen Ctrl+Alt+Enter
Unity
Manage
Help
Exit
CD/DVD (IDE)
Floppy
Network Adapter
Printer
Serial Port 2
Sound Card
Fitipower Integrated Mass Storage Device
Gemplus USB SmartCard Reader
Anchor Chips OSR USB-FX2 LK
Shared Gemplus USB Smart Card Reader 0
Connect (Disconnect from host)
Change Icon...
Show Icon
BasicUsb

The Basic USB driver comes with its own inf file that creates a setup class for OSR Basic USB Devices. Once you have connected your OSR USB FX2 device, the driver can be installed with the following devcon command (note both the command and the quotes, which are part of the syntax):

```
devcon update basicusb.inf "USB\Vid_0547&Pid_1002"
```

*Read the above carefully. Note that the devcon command here is "**update**" – NOT "install" as it was for the Nothing driver. This is because there's real hardware involved, and devcon's syntax is... unusual.*

If using Device Manager is more your style, you can connect the OSR USB FX2 board, right click its device entry in Device Manager, select "update driver" and point it to a directory on your target system where you have copied the "driver package" (sys file, INF file, and CAT file) from your device system.

Once you've installed the driver once, you can update it in the usual way by just copying an updated version of the SYS file over the old one in the \windows\system32\drivers directory. Note that you might have to disable the device in Device Manager before Windows will allow you to overwrite the old driver file.

Lab three examines the basic mechanics of communicating with a USB device from a KMDF USB client driver, KMDF continuous readers, and manual queues.

Notes on the OSR USB FX2 Device

*The bulk pipes in the OSR USB FX2 device work in a loopback configuration, so data sent to the bulk **out** pipe will be read back from the bulk **in** pipe.*

*If the device's internal data buffers are full when a bulk **out** operation is requested, the operation is pended by the bus driver until room is available. If the device's internal data buffers are empty when a bulk **in** operation is requested, the operation is pended by the bus driver until data arrives.*

More detailed information on the OSR USB FX2 device is provided within the OSR USB FX2 Data Sheet. This document immediately follows the last lab exercise in this handout. If you haven't read it yet, you really DO need to read it before you do these assignments.

Assignment 5A

The Basic USB code as supplied meaningfully the EvtIoWrite Event Processing Callback, converting write requests into bulk out requests and sending them to the USB bus driver. When installing this version of the driver, you should be able to successfully perform one or more write operations using the basicusbtest application before the writes start to hang due to the device's internal buffers being full (again, see the OSR USB FX2 Data Sheet for why this is the case).

Modify the driver to implement a read handler that converts the read requests into bulk in requests and sends them to the USB bus driver.

To test your changes, perform a read operation with the basicusbtest application. Assuming you have performed no previous writes to the device, this read should pend, waiting for the I/O to complete. Bring up a second instance of the basicusbtest application and perform a write operation. The read operation in the first instance, and the write operation, should both complete with success.

Hints:

1. You'll need to get the WDFUSBPIPE Target for the device's bulk in pipe.
2. The buffer associated with a read request is the "**output buffer**".

Assignment 5B

Add a new IOCTL (perhaps named IOCTL_OSR_BASICUSB_GET_SWITCHPACK_STATE) that gets last known state of the switch pack. Modify the basicusbtest application to test your changes.

There are multiple ways to implement this task. But if you read the provide basicusb sample code, you'll see that it already implements a continuous reader on the interrupt pipe. At startup, and each time the switch pack state on the OSR USB FX2 device changes, the reader returns the state of the switch pack. So, it's probably easiest to use this existing code. Don't worry if this seems too simple for you. We'll get more complicated in later assignments of this exercise.

Hints:

1. You'll need to keep track of the current switch pack state somewhere.

Assignment 5C

Take the IOCTL from 5B and, instead of completing it immediately, complete it when the state of the switch pack changes. Also, return to the user the new state of the switch pack.

Hints:

1. For the purposes of this assignments, you should ignore the fact that the switch pack isn’t debounced by the OSR USB FX2 device.

2. You'll need someplace to store the incoming user requests that are waiting for the switch pack state change.

(This page intentionally left blank)

Lab Exercise 6 – File Objects, Open and Close

In this assignment, you'll get some experience implementing File Object related Event Processing Callbacks for Create and Close. You'll also get more experience processing read and write requests.

Use the driver you created in Exercise 3 (which was based on the Nothing_KMDF Driver) as the basis for this assignment.

Assignment 6A

Modify the driver you created in Assignment 3B to only allow Write requests from the first opener of the device. That is, the only write Requests your driver should complete successfully are those associated with the first open handle on the device. Complete all other write requests with STATUS_ACCESS_DENIED. Note that reads must be allowed from any requestor.

If you want to get fancier, and if you made it this far successfully and are still doing assignments, we suggest you keep track of the order of open instances. When the first opener closes the device, "promote" the next opener to be allowed to write.

Hints:

1. Ask yourself why you're still reading the hints this far into the assignments. C'mon, you can do this lab assignment without using any hints!

2. Well, it **is** a pretty hard assignment, and you probably deserve a couple of hints. Which data structure uniquely identifies open instances of a file or device? A user application gets a handle to this internal Windows data structure when they open a file or device.

3. Recall that Open and Close are File Object related methods in WDF and are established using the WDF_FILEOBJECT_CONFIG_INIT macro.

4. You'll need to implement EvtFileCreate and EvtFileClose Event Processing Callbacks. In EvtFileCreate you'll need to keep track of the number of open instances and store the File Object that represents the first open instance of your device.

5. In terms of tracking File Objects and handling the "promotion" part of the assignment, consider using a WDFCOLLECTION to organize the File Objects that represent open instances of your device.

(This page intentionally left blank)

Setting Up a VMWare Virtual Machine for Use with WinDbg

Introduction

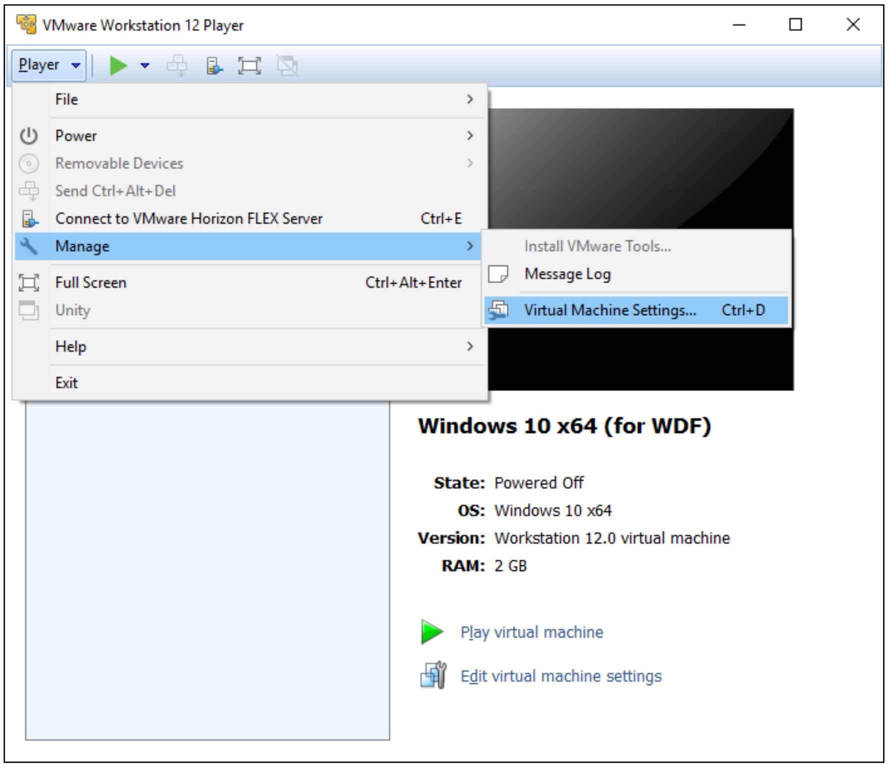
It’s easy to setup a VM as a target machine… after you’ve done it once. Here’s a step-by-step guide to making that first time easy. There are only 10 steps.

The steps DO vary SLIGHTLY, depending on whether you’re using VMware Player or VMware Workstation. They also tend to vary a bit depending on the version of VMware that you’re using. We’ll use VMware Player 12 in our example, but regardless of which program or which version you use, these steps should be sufficient to guide you through the process.

Steps

Step 1

Start VMware Player and open the Virtual Machine you’re going to use as a Target system (Player->File->Open). This machine must be fully powered-off (not simply suspended). Once the VM has been opened, from the Player menu select Manage->Virtual Machine Settings:

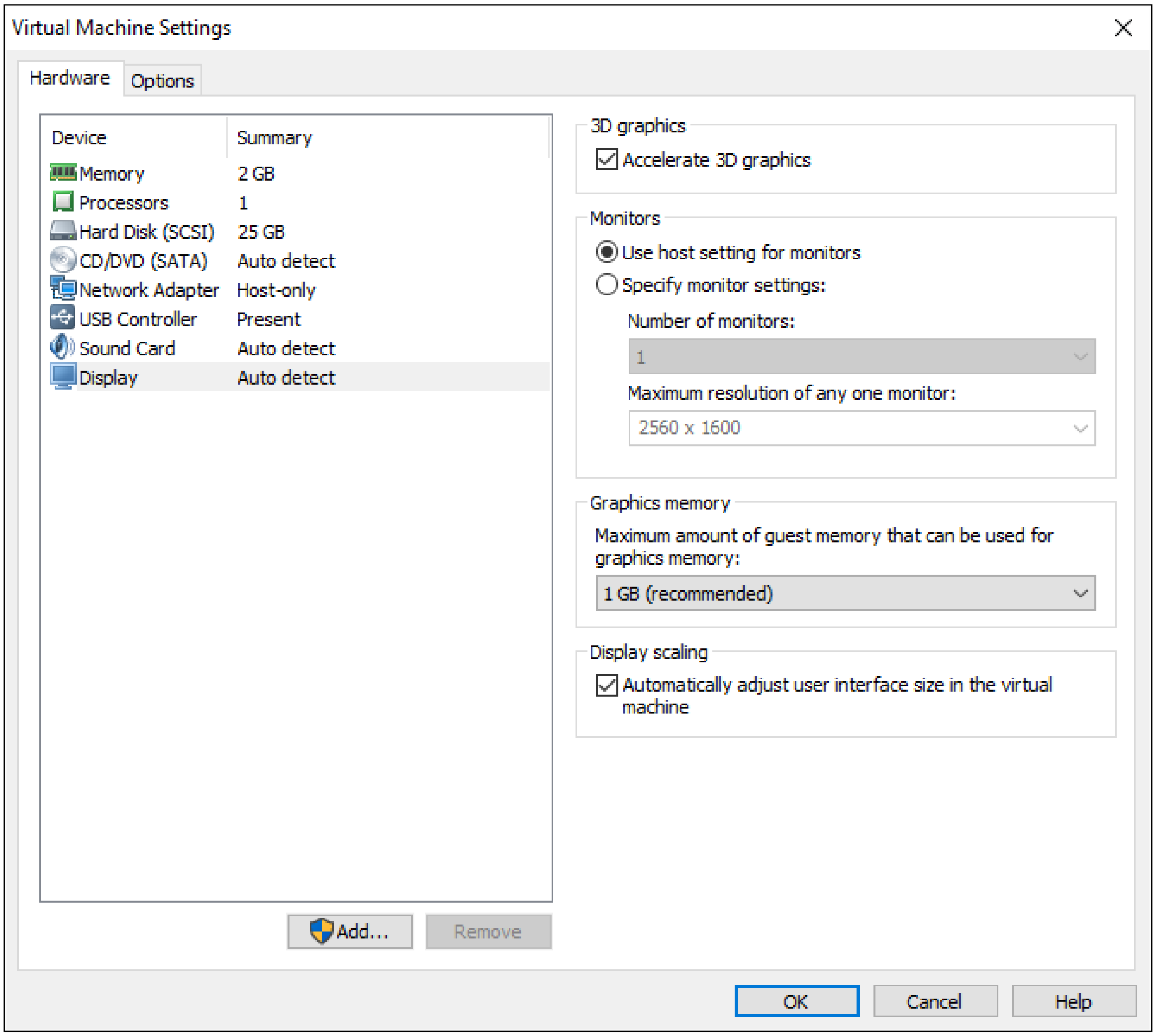

VMware Workstation 12 Player
Player
File
Power
Removable Devices
Send Ctrl+Alt+Del
Connect to VMware Horizon FLEX Server Ctrl+E
Manage
Full Screen Ctrl+Alt+Enter
Unity
Help
Exit
Install VMware Tools...
Message Log
Virtual Machine Settings... Ctrl+D
Play virtual machine
Edit virtual machine settings

Windows 10 x64 (for WDF)

State: Powered Off
OS: Windows 10 x64
Version: Workstation 12.0 virtual machine
RAM: 2 GB

Step 2

If the Virtual Machine is not already setup with a Serial Port (sometimes we take care of this for you when we build the VMs for class). Click the "Add…" button:



Device	Summary
Memory	2 GB
Processors	1
Hard Disk (SCSI)	25 GB
CD/DVD (SATA)	Auto detect
Network Adapter	Host-only
USB Controller	Present
Sound Card	Auto detect
Display	Auto detect

3D graphics

☑ Accelerate 3D graphics

Monitors

● Use host setting for monitors

○ Specify monitor settings:

Number of monitors: 1

Maximum resolution of any one monitor: 2560 x 1600

Graphics memory

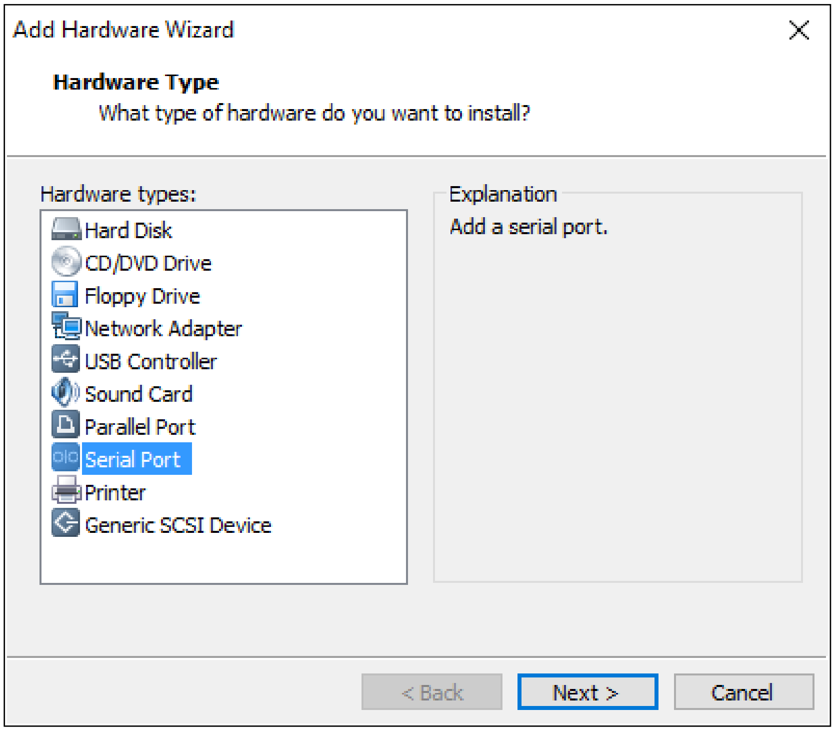
Maximum amount of guest memory that can be used for graphics memory: 1 GB (recommended)

Display scaling

☑ Automatically adjust user interface size in the virtual machine

Step 3

The Add Hardware Wizard dialog box appears. Select "Serial Port" from the Hardware list on the left, and click "Next >":



Add Hardware Wizard

Hardware Type

What type of hardware do you want to install?

Hardware types:

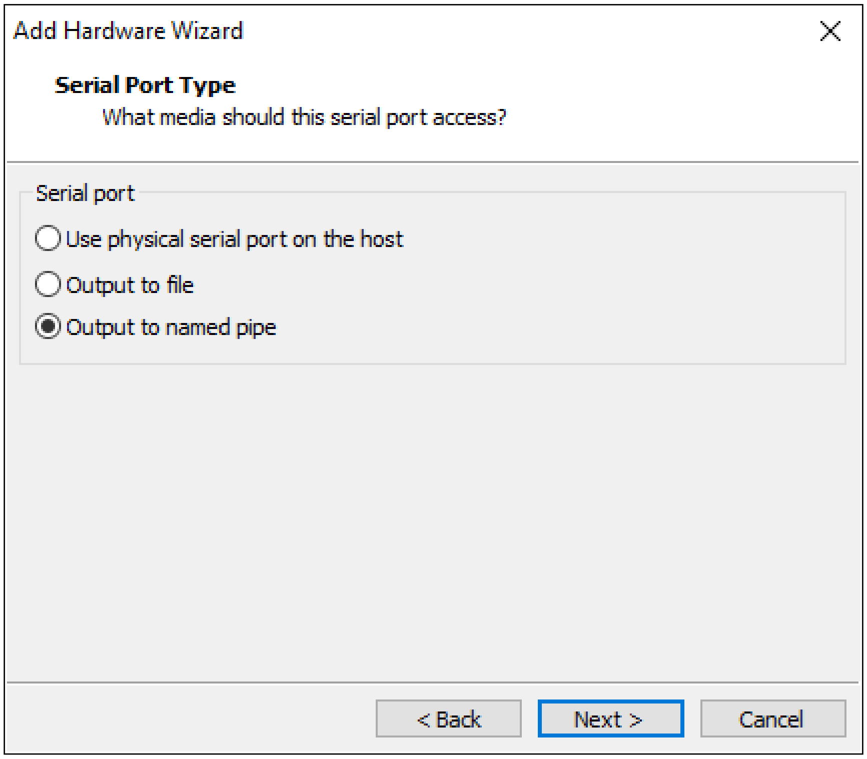
- Hard Disk
- CD/DVD Drive
- Floppy Drive
- Network Adapter
- USB Controller
- Sound Card
- Parallel Port
- Serial Port
- Printer
- Generic SCSI Device

Explanation

Add a serial port.

Step 4

Indicate that the serial port you’ll be using should access a named pipe:



Add Hardware Wizard

Serial Port Type

What media should this serial port access?

Serial port

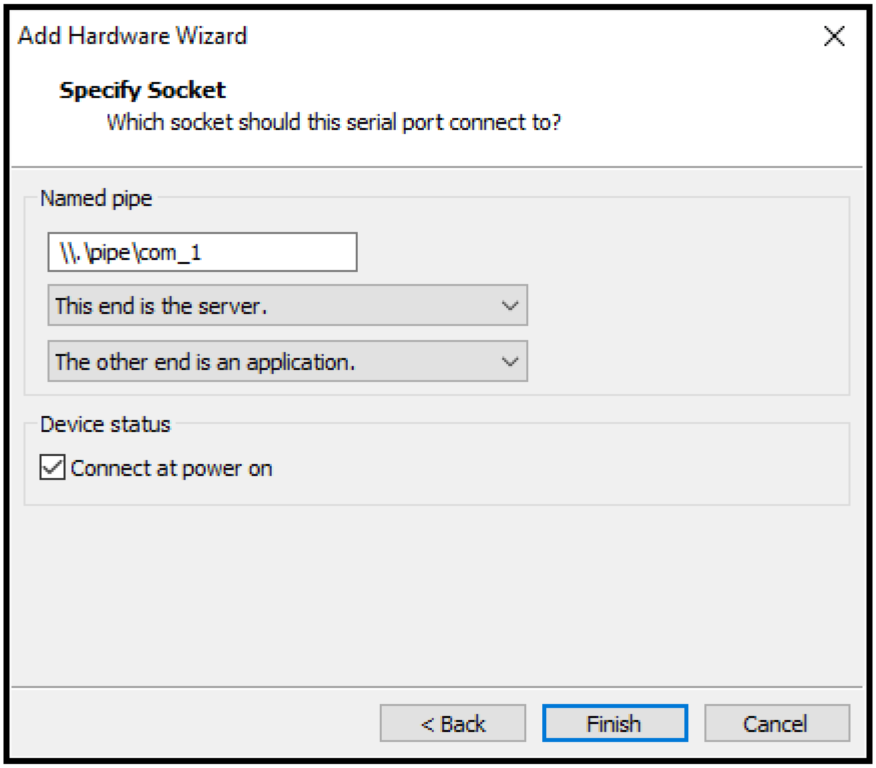
○ Use physical serial port on the host

○ Output to file

● Output to named pipe

Step 5

Indicate the name of the named pipe and the settings shown below. Carefully match ALL these settings. You will have to change to the "The other end is an application" setting from the default, as shown below:



Add Hardware Wizard

Specify Socket

Which socket should this serial port connect to?

Named pipe

\\.\pipe\com_1

This end is the server.

The other end is an application.

Device status

☑ Connect at power on

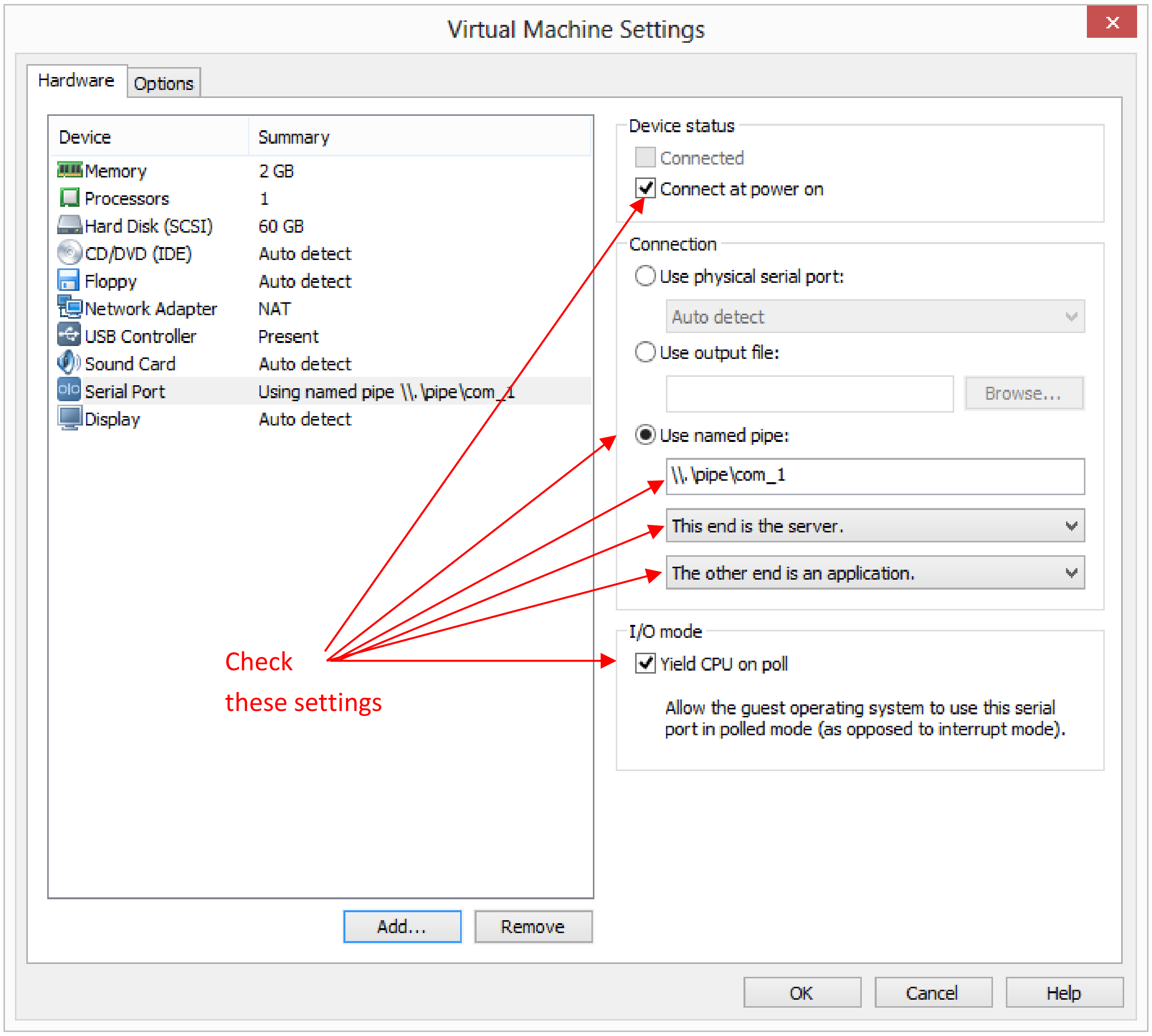
Click "Finish".

Step 6

You should once again see the Virtual Machine Settings dialog box, but now, a serial port has been added. Verify that the settings are what you expect. Fix anything that's wrong.

IMPORTANT NOTE: Be sure that the "Yield CPU on poll" check box is checked. If it's not, check it now.

When you're satisfied with the settings, click "OK":



Virtual Machine Settings

Hardware | Options

Device	Summary
Memory	2 GB
Processors	1
Hard Disk (SCSI)	60 GB
CD/DVD (IDE)	Auto detect
Floppy	Auto detect
Network Adapter	NAT
USB Controller	Present
Sound Card	Auto detect
Serial Port	Using named pipe \\.\pipe\com_1
Display	Auto detect

Device status

□ Connected

☑ Connect at power on

Connection

○ Use physical serial port:

Auto detect

○ Use output file:

● Use named pipe:

\\.\pipe\com_1

This end is the server.

The other end is an application.

I/O mode

☑ Yield CPU on poll

Allow the guest operating system to use this serial port in polled mode (as opposed to interrupt mode).

Check these settings

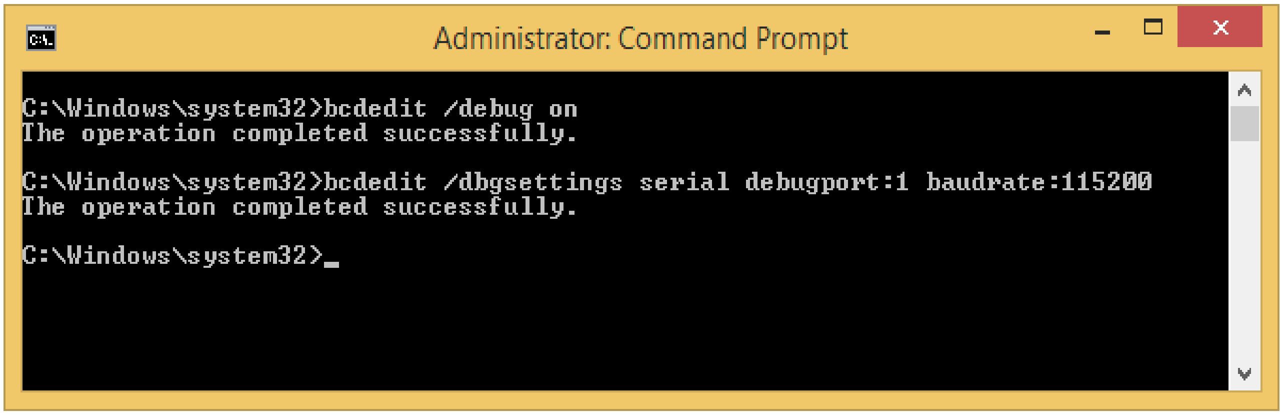
Step 7

Start the VM and wait for Windows to start. On the Target machine running in the VM open an **administrator** command prompt window (don’t forget to “run as administrator”) in the VM and enable support for kernel debugging in the Target system using the following commands:

bcdedit /debug on

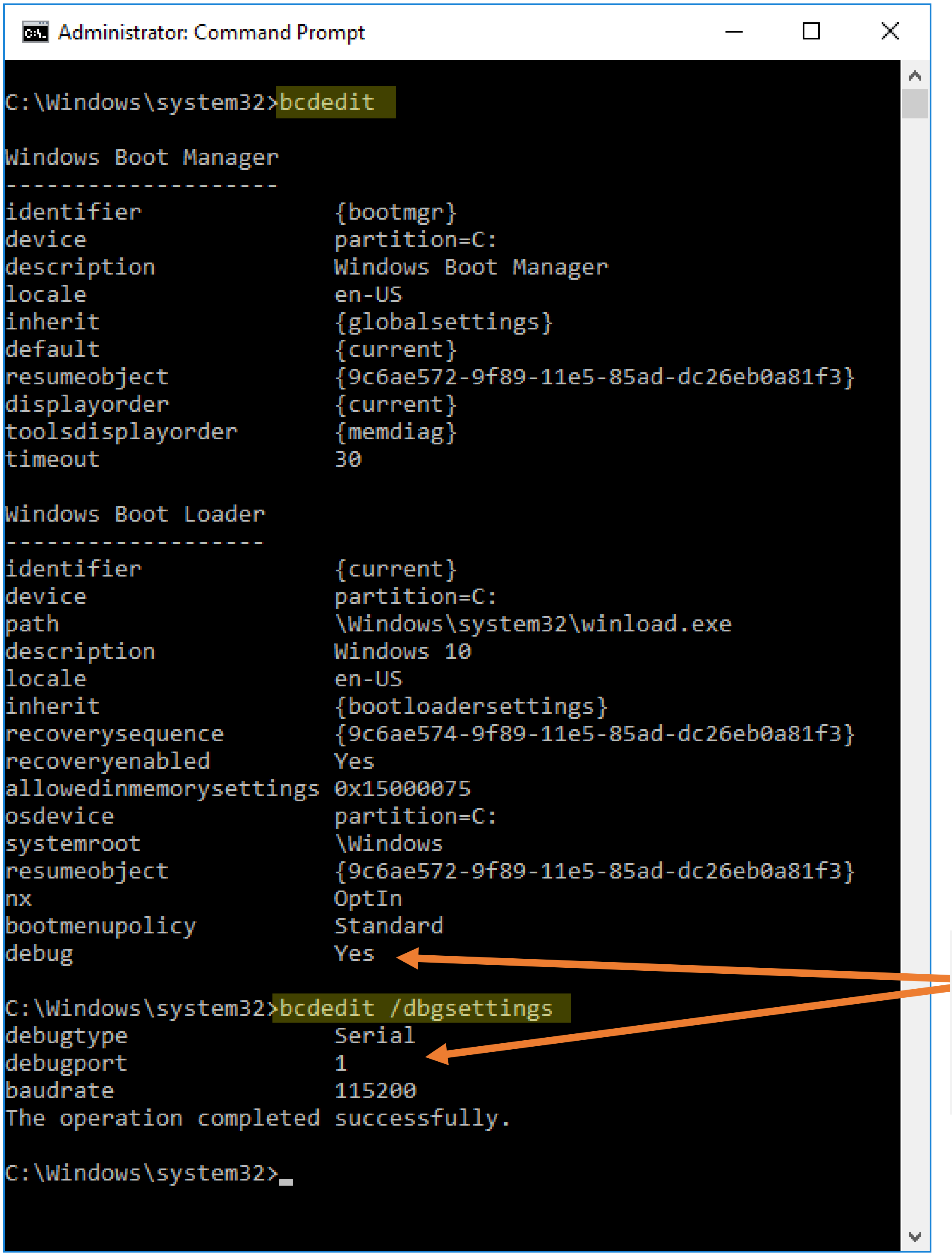
bcdedit /dbgsettings serial debugport:1 baudrate:115200

Like this:



```
C:\Windows\system32>bcdedit /debug on
The operation completed successfully.

C:\Windows\system32>bcdedit /dbgsettings serial debugport:1 baudrate:115200
The operation completed successfully.

C:\Windows\system32>_
```

You can check to see if debugging is enabled, by issuing just the command "bcdedit" (with no parameters). You should see the parameter "Debug" set to on. You can check to see if the debugger configuration settings match what you supplied by issuing the command "bcdedit /dbgsettings" (with no additional parameters). You should see the debugger connection method, port, and speed you specified. Your output should look like that shown below:



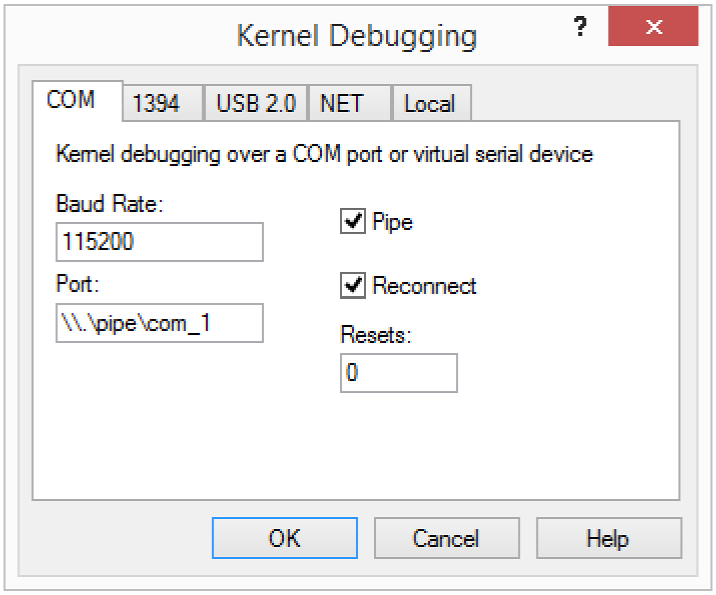
```
C:\Windows\system32>bcdedit

Windows Boot Manager
--------------------
identifier              {bootmgr}
device                  partition=C:
description             Windows Boot Manager
locale                  en-US
inherit                 {globalsettings}
default                 {current}
resumeobject            {9c6ae572-9f89-11e5-85ad-dc26eb0a81f3}
displayorder            {current}
toolsdisplayorder       {memdiag}
timeout                 30

Windows Boot Loader
-------------------
identifier              {current}
device                  partition=C:
path                    \Windows\system32\winload.exe
description             Windows 10
locale                  en-US
inherit                 {bootloadersettings}
recoverysequence        {9c6ae574-9f89-11e5-85ad-dc26eb0a81f3}
recoveryenabled         Yes
allowedinmemorysettings 0x15000075
osdevice                partition=C:
systemroot              \Windows
resumeobject            {9c6ae572-9f89-11e5-85ad-dc26eb0a81f3}
nx                      OptIn
bootmenupolicy          Standard
debug                   Yes

C:\Windows\system32>bcdedit /dbgsettings
debugtype               Serial
debugport               1
baudrate                115200
The operation completed successfully.

C:\Windows\system32>_
```

Check your settings… Do they match what you specified?

Step 8

Back on your host machine, start WinDbg. Select "File" from the program menu and select "Kernel Debug…" from the drop down. The Kernel Debugging dialog will appear.

In the Kernel Debugging dialog, ensure the Pipe and Reconnect check boxes are checked, and set the name of the port as shown below to match the name of the port you specified in your Virtual Machine. Click "OK" when done:



Kernel debugging over a COM port or virtual serial device

Baud Rate: 115200

Port: \\.\pipe\com_1

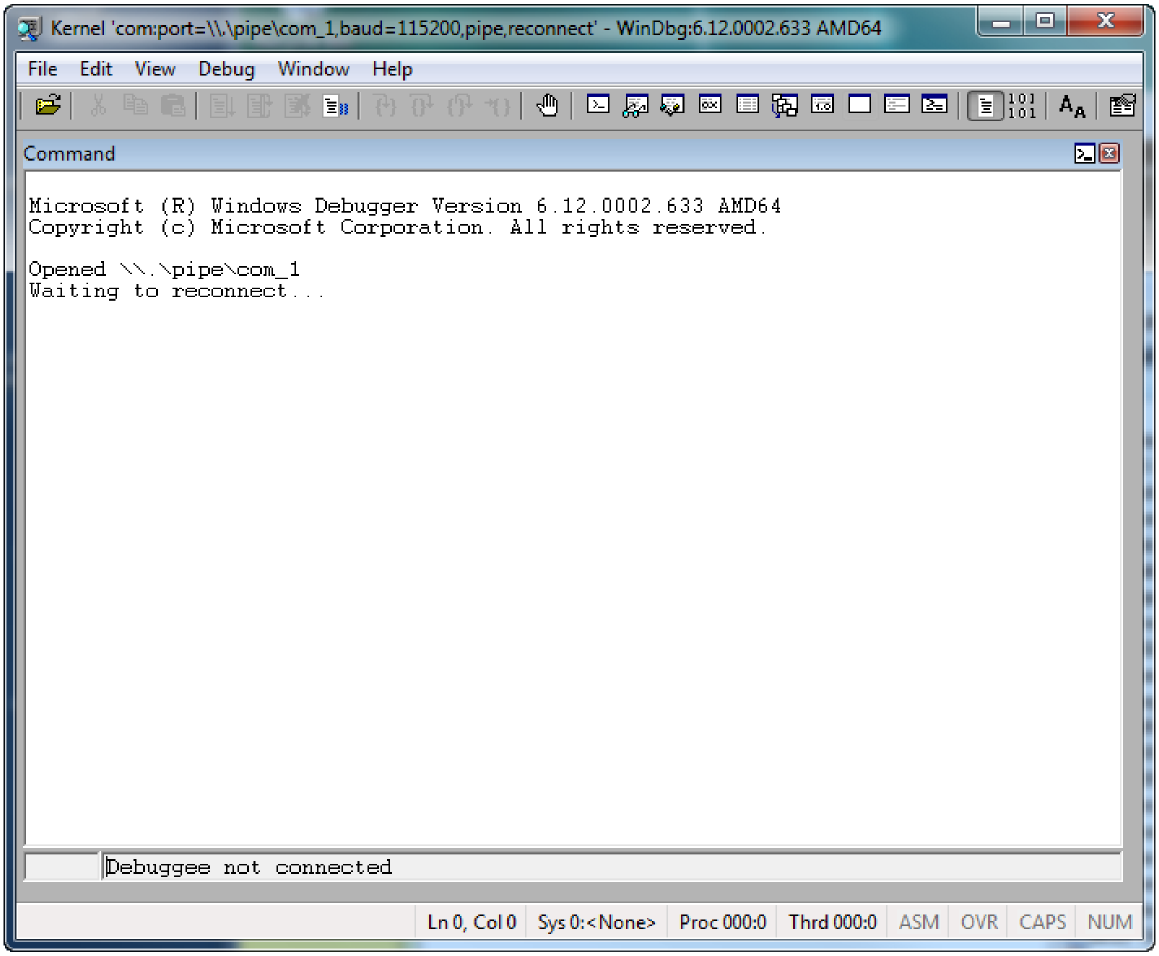
☑ Pipe

☑ Reconnect

Resets: 0

Step 9

After clicking "OK" in the Kernel Debugging dialog box, WinDbg will be waiting for a connection to your Target machine as follows:

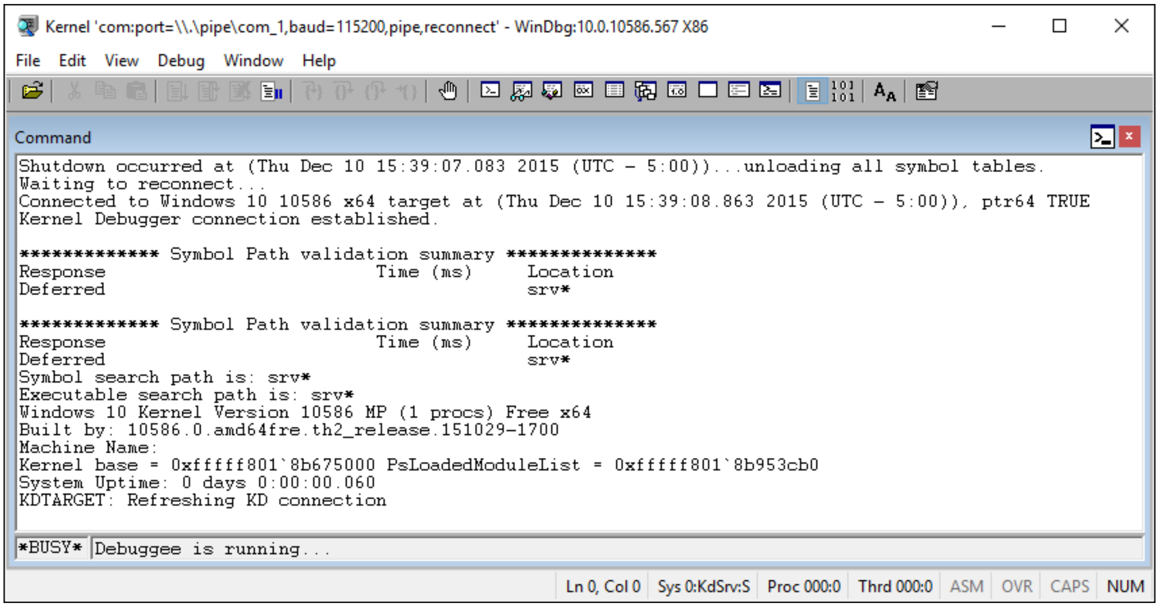

Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe,reconnect' - WinDbg:6.12.0002.633 AMD64
File Edit View Debug Window Help
Command
Ln 0, Col 0
Sys 0:<None>
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM

```
Microsoft (R) Windows Debugger Version 6.12.0002.633 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\pipe\com_1
Waiting to reconnect...
```

```
Debuggee not connected
```

Step 10

Now that WinDbg is set up on your host machine, reboot the target machine. As the target machine reboots, you should see output in WinDbg like the following:



Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe,reconnect' - WinDbg:10.0.10586.567 X86
File
Edit
View
Debug
Window
Help
Command

```
Shutdown occurred at (Thu Dec 10 15:39:07.083 2015 (UTC - 5:00))...unloading all symbol tables.
Waiting to reconnect...
Connected to Windows 10 10586 x64 target at (Thu Dec 10 15:39:08.863 2015 (UTC - 5:00)), ptr64 TRUE
Kernel Debugger connection established.

************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*

************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*
Symbol search path is: srv*
Executable search path is: srv*
Windows 10 Kernel Version 10586 MP (1 procs) Free x64
Built by: 10586.0.amd64fre.th2_release.151029-1700
Machine Name:
Kernel base = 0xfffff801`8b675000 PsLoadedModuleList = 0xfffff801`8b953cb0
System Uptime: 0 days 0:00:00.060
KDTARGET: Refreshing KD connection
```

```
*BUSY* Debuggee is running...
```

(This page intentionally left blank)

Using the OSR USB FX-2 Learning Kit V2.0

Board Rev: 00
Firmware Revision: 3.5
Document Revision: 2.1

This document describes how to use the OSR USB FX-2 Learning Kit, V2.0. This USB device was designed and built by OSR specifically for use in teaching software developers how to write drivers for USB devices. This document describes the basic functionality available on the board. The target audience for this document is Windows device driver writers who want to use or implement a driver for the board.

Configurations

The board is configured with the following VID, PID, and version:

Parameter	Value	Assigned To
Vendor ID	0x0547	Cypress Semiconductor (actually Anchor Chips, Inc).
Product ID	0x1002	Sample device
Version ID	0	

The vendor and product IDs used by the board are the same as those used by the development board supplied with the Cypress EZ-USB FX2 Development Kit (CY3681).

The board supports a single configuration. The board automatically detects the speed of the host controller and supplies either the high or full speed configuration based on the host controller’s speed. The alternate speed is also supplied as an “other speed configuration descriptor.”

Interfaces

In both high-speed and full-speed mode, the board implements Interface 0.

Endpoints

The firmware supports 3 endpoints, as follows:

Endpoint Number	EP Type and Use	Packet Size in FULL SPEED	Packet Size in HIGH SPEED
1	Interrupt, IN from board to host	1 byte	1 byte
6	Bulk, OUT from host to board	64	512
8	Bulk, IN from board to host	64	512

Endpoint number 1 is used to indicate the state of the 8-switch switch-pack on the OSR USB-FX2 board. A single byte representing the switch state is sent (a) when the board is first stated, (b) when the board resumes after selective-suspend, (c) whenever the state of the switches is changed.
Endpoints 6 and 8 perform an internal loop-back function. Data that is sent to the board at EP6 is returned to the host on EP8.

Additional information about the how each endpoint works is provided in the following sections.

Endpoint 1 -- Interrupt Endpoint Packet Format

The board sends an 8-bit value that represents the state of the switches on the switch pack from EP1. No more than 1 byte of data is ever sent at one time. The switch pack value is sent on startup, resume from suspend, and whenever the switch pack setting changes. Note that the firmware *does not de-bounce* the switch pack, so unless the switches are very quickly and firmly switched, one switch change can result in multiple bytes being sent.

Note that the bits indicating the switches are backwards with respect to the numbers on the switch pack. That is, the value that’s sent in bit 0 by the firmware reflects the switch that’s labeled “8” on the switch pack.

Endpoint one is single buffered. That is, the firmware will buffer the information from exactly one switch pack transition, and the switch pack’s state is not queried again until the buffered transition data is read by the client driver.

The best strategy to use in implementing client driver support for this feature is to always have a small number of reads for EP1 data queued.

Endpoint 6 and 8 Bulk Packet Format

The board receives data on EP6 and sends the same data back on EP8. The board does not change the values of the data that it receives on EP6 and does not internally create any data on EP8. Thus, the board performs a pure EP6 to EP8 loop-back operation.

The maximum packet size accommodated by EP6 and EP8 depends on the speed at which the board is operating. When operating in full speed, the maximum packet size is 64 bytes. When operating in high speed, the maximum packet size is 512 bytes.

Endpoint 6 and endpoint 8 are always double buffered. This means, for example, that up to 4 data packets (of any size, up to the maximum packet size) can be sent to EP6 before any data is read on EP8. The buffering arrangement that makes this possible is illustrated in Figure 1.

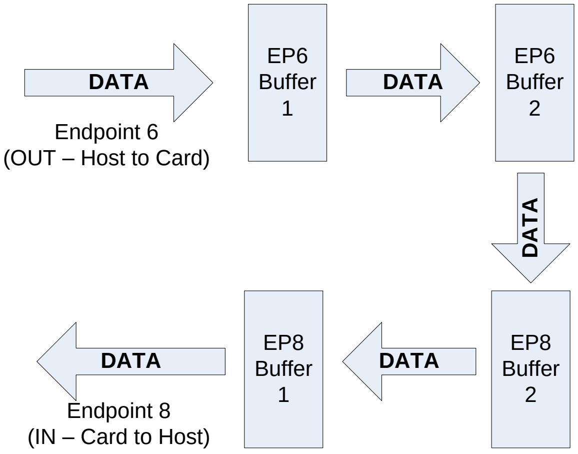


FIGURE 1 -- HOW EP6 AND EP8 BUFFERS ARE USED

If you send 4 data packets to EP6 without issuing any reads to EP8, the first two data packets sent to EP6 will be accommodated by EP8’s output buffers, and the last two data packets sent to EP6 will be buffered in EP6’s input buffers. If you attempt to send a 5th data packet from the client driver to EP6 without sending any reads to EP8, the write request will wait in the Windows host (bus) driver until EP6 buffer space is available on the board. This buffering scheme applies regardless of the number of bytes that you send per packet. Even a packet that’s one byte long will consume one complete endpoint buffer.

Continuing with this example, if you have sent 4 data packets to EP6, and then you issue a read to EP8, the contents of the first data packet sent to EP6 will be returned on EP8. As a result of the read to EP8, one endpoint buffer is again available on EP6.

Board LED Displays

The OSR FX2 firmware uses the LED displays on the board as described in this section.

7-Segment LED display

During initialization, the board displays the firmware major and minor revision numbers in the LED, 1 digit at a time, with a 500ms wait between digits.

After the firmware revision has been displayed, the firmware puts an “F” or “H” in the display to indicate whether the board firmware is running in full or high-speed mode, respectively.

Whenever the device is put into a selective suspend state by the USB Driver, the firmware will display a “S” in the 7-Segment display. When the board is returned to active state from suspended state, the firmware displays an “A” (for “active”) in the 7-segment display.

The 7-Segment display can also be set or read by the client driver using vendor-commands described later in this document.

LED Bar Graph Display

The LED bar graph display is used by default to indicate bulk packet activity. One light is lit on the LED bar graph for each 10 bulk packets that the board transmits. From 0 to 8 LED segments are displayed (note that top 2 LED segments are never lit, because they're not connected to anything on the board). After all 8 LEDs have been lit, the display wraps-around to no LEDs being lit.

The contents of the bar graph can also be set or read by the client driver using vendor commands described later in this document.

Vendor Commands

The firmware supports several vendor specific commands which allow a driver to query the state of LEDs and switches as well as modify the state of LEDs.

Below, Figure 2 gives a general layout of the device, its various LEDs, Switches and Segmented LEDs. The reader should note the numbering on the various LEDs, Switches and Segmented LED because the input and returned bitmasks represent their placement on the device.

Three cases are specifically worth noting:

- The numbering of the LEDs in the bar graph block is not intuitive. When reading the state of the LEDs a returned value of 0x08, indicates that “LED 1” is on, whereas a returned value of 0x10, indicates that “LED 2” is on. The numbering shown in Figure 2 should make this clear. In case you didn't see it mentioned above, the top to LEDs in the bar graph aren't connected to anything... so you won't be able to light them.

- The switch numbers, as addressed by the device firmware, are reversed from the number printed on the switch pack on the board. For example, a returned value of 0x01 indicates that Switch 1 is toggled, even though it is numbered “8” on the switch pack on the board. Again, the numbering shown in Figure 2 shows the actual number used by the firmware.

- For the 7-segment display, the hex values shown in Figure 2 correspond to the value to be passed to the board to light the indicated segment of the display. For example, to display a "1" in the display, you'd want to light the two right-most vertical display segments. To light these two segments, send the value 0x6, which is 0x02 (to light the top right-most vertical bar) OR'ed with 0x04 (to light the bottom right-most vertical bar). If you're wondering what the funny little block with the value 0x08 is on diagram, it's for the decimal point. So, actually, it's an 8-segment display, isn't it.

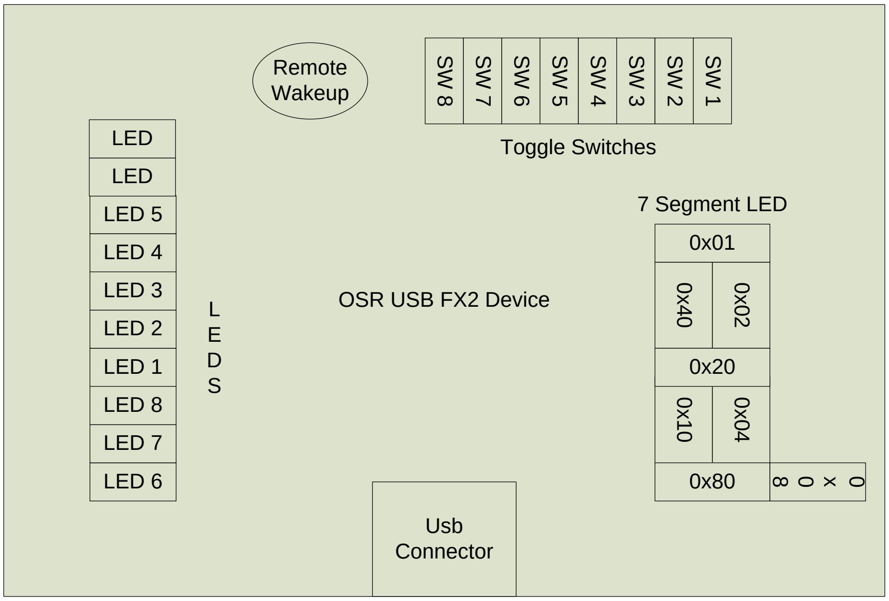


Figure 2 - OSR FX2 Device Layout

The Vendor commands are built, by the driver, using the **UsbBuildVendorRequest** routine. The routine is defined as follows:

```
VOID
UsbBuildVendorRequest(
IN PURB    Urb,
IN USHORT  Function,
IN USHORT  Length,
IN ULONG   TransferFlags,
IN UCHAR   ReservedBits,
IN UCHAR   Request,
IN USHORT  Value,
IN USHORT  Index,
IN PVOID   TransferBuffer  OPTIONAL,
IN PMDL    TransferBufferMDL  OPTIONAL,
IN ULONG   TransferBufferLength,
IN PURB    Link  OPTIONAL,
);
```

All vendor commands are issued with the USB function **URB_FUNCTION_VENDOR_DEVICE**, the commands and their requirements are listed below:

Vendor Command	Driver Input Requirements	Driver Output
0xD4 – READ 7 SEGMENT DISPLAY Placed in Request Field. Reads the current state of the 7-segment display	None.	Transfer Buffer = segmented LED bits, returned as a UCHAR value TransferBufferLength = size of Buffer containing State
0xD6 – READ SWITCHES Placed in Request Field. Reads the state of all switches on the board and returns a state bitmask.	None.	Transfer Buffer = switch bitmask, returned as a UCHAR value. TransferBufferLength = size of Buffer containing State
0xD7 – READ BARGRAPH DISPLAY Placed in Request Field. Reads the state of the LEDs in the bargraph display on the board and returns a state bitmask.	None.	Transfer Buffer = LED bitmask, returned as a UCHAR value. TransferBufferLength = size of Buffer containing State
0xD8 – SET BARGRAPH DISPLAY Placed in Request Field. Sets the state of all LEDs on the board via an input bitmask.	Value = LED BitMask (UCHAR Value) Bits 0 – 7, corresponding to segments of display to light. See Figure 1.	None.
0xD9 – IS HIGH SPEED Placed in Request Field. Returns the state of the device, for use on early Win2K systems that do not support bus driver's standard feature.	None.	Transfer Buffer = Speed State: 1 = HighSpeed, 0 = Full Speed, returned as a UCHAR value. TransferBufferLength = size of Buffer containing State
0xDB – SET 7 SEGMENT DISPLAY Placed in Request Field. Sets the state of the 7-segment LED display on the board via an input bitmask.	Value = LED BitMask, (UCHAR Value) Bits 0 – 7, corresponding to segments to light (see Figure 1).	None.

Advanced Usage Information

This section describes advanced information about how the board can be used. You do not have to read this section if you are only interested in knowing the information that's required to write a driver for the OSR USB FX2.

The OSR USB FX2 device is loosely based on the development board supplied with the Cypress EZ-USB FX2 Development Kit (CY3681). You can optionally use the Cypress demo driver EZUSBDRV.SYS to control the OSR USB FX2 board, because the board uses the Cypress USB FX2 chipset and the OSR board's vendor ID and product ID are identical to that of the Cypress development board.

> **Important Note**
> **Using Cypress EZUSBDRV.SYS and Other Cypress Software**
>
> OSR does not endorse, nor do we recommend, that you use the Cypress demonstration driver with this board. Our experience with the EZUSBFX2 driver has not been good, and we have experienced numerous system crashes when using it.
>
> OSR does not provide support for, or assistance with using, any Cypress software including the EZUSBDRV or its associated applications.

Because the board is compatible with the Cypress USB driver, you can also use it with the test applications provided in the Cypress EX-USB Development Kit. This includes the Cypress EZ-USB control panel application, and the "bulk loop" sample application from the kit.

The OSR USB FX2 Learning Kit was specifically designed to allow different firmware to be downloaded to the device. If you're using the Keil uVision tools to build firmware for the device, we recommend setting the Code Range option to 0x80-0xFFF and the Xdata Range option to 0x1000 in the BL51 Locates tab of the Target Options dialog.

The OSR USB FX2 device does not support in-circuit reprogramming of the EEPROM. However, because the EEPROM is socketed, you can easily replace the EEPROM with one that you program yourself using an EPROM programmer.

Please do not even think about using the OSR USB FX2 board as a reference for any sort of hardware design. We built this board with no other goals in mind other than (a) creating a board that works for the purposes of learning how to write Windows drivers and (b) keeping the board as inexpensive as possible. In so doing, we've violated many, many, of Cypress' design recommendations.

Device Problems or Questions

OSR Open Systems Resources, Inc. warrants that this board will be free of defects in materials or workmanship for a period of 30 days from purchase. We know that's not very long, but this is a development/test board, and we're selling at our cost.

Answers to questions about how to use this board that aren't answered in this document can be directed to OSR via email to hwsupport@osr.com.

General questions about how to write device drivers, or how to debug a driver you're writing, should be directed to the NTDEV list that OSR administers. Visit the list server section of www.osronline.com to join the list or search the archives.

We don't want to sound cruel, BUT: Unless you're one of our current or former students (who are *always* welcome to send us questions on *any* topic), please don't send us random questions asking us to help you write or debug your driver. There are thousands of you out there and only a few of us, so we can’t help everybody. Please use the NTDEV list. You’ll find people on that list who know more about USB drivers than we do, anyways.

Frequently Asked Questions

Since release of this device, we’ve gotten many questions sent to hwsupport@osr.com, and we see some of them frequently enough that they qualify for inclusion here:

Q: *Will you guys send me the source code to the firmware on the device?*
A: No. It’s not hugely different from the FX2 BULKLOOP sample in the Cypress Development Kit. It’s not that we’re trying to be difficult, but we just didn’t write the code for external consumption, and we’d have to take all the bad words out of it first.

Q: *PLEASE?*
A: Sorry. No, we will not accept money (a rare statement from anyone here at OSR).

Q: *Will you guys send me the schematic for the device or explain the pinouts from the chip? It’s perfect for what I need, except for mumble.*
A: No. We didn’t create this device as a cheap alternative to the Cypress USB-FX2 Development Kit. We created it as a cheap way for folks to learn how to write Windows drivers. Anyhow, if you can’t figure out what pins connect to what from looking at a two-sided board and reading the chip spec sheet, you really don’t have any business building your own hardware.

Q: *Do you have a Linux driver for the board?*
A: You want a Linux driver, call the guy in Finland. But seriously, tons of people **have** told us that they've written Linux drivers for the board. Just not us and, no, we don't remember who they are so please don't ask.

Q: *Is there someplace I can download a Windows WDF/KMDF driver for the board?*
A: No. The example KMDF driver is in the WDK.

Q: *I’d like to buy xxx boards for the folks I work with. Can I get a volume discount?*
A: Did you miss the part about us selling the boards *at our cost*? So, if we were to give you a volume discount, it would cost us money. We might be able to work something out on shipping, though, if you want us to ship them via ground transportation and in one big box. One of our clients has purchased close to 200 of these boards so far, and they pay the price listed on the web site with an adjustment for shipping. We’ll do the same for you if you’d like to buy, say, 25 or more boards at one time.

Q: *Hey, you guys screwed up! The bits on the switch pack are numbered backwards relative to the way they're reported by the firmware.*
A: Actually, this is intentional. Drivers are frequently called-on to "fix up" things that aren't perfect in the hardware. We didn't want to build a device that was artificially simple to use.

Q: *Are you guys too dumb to write firmware to de-bounce the switch pack?*
A: Yes, obviously, that's it. Please read the answer to the previous question.

Differences Between OSR USBFX-2 V1.0 and V2.0

The kits are functionally identical and are programmed the same way. The major changes we made between V1.0 and V2.0 were (a) we change the board layout to use surface-mount components wherever practical, (b) we added more copper to serve as a slightly better ground plane, (c) we change the BOM to make the board ROHS compliant.

OSR Thanks….

If you're using this board, you owe a major debt of thanks to John Nesselhauf, who donated his valuable, expert, time designing this board and laying it out, to make the board a reality. Thanks also go to (in alphabetical order) Randy Aull, Doron Holan, Nar Ganapathy, and Eliyas Yakub for feedback, ideas, and encouragement. And, as always, OSR remains grateful to its engineering team, without whose enthusiasm this project would have died. Again.